

Parallel Parsing Processes *Revisited*

Michael Spivey
University of Oxford



Department of
COMPUTER
SCIENCE



Thompson [1968]

Compiles regexps into NFAs represented as machine code (for the IBM 7094).

Matching machine reads the input one character at a time, and dynamically maintains two lists of subroutine calls:

- *CLIST* – alternatives for the current character
- *NLIST* – alternatives for the next character.

[Makes a great student project!]

[1] Ken Thompson, "Programming Techniques: Regular expression search algorithm," CACM 11, 6 (June 1968), pp. 419--22.

Translating regexps

- For c : if $char = c$ then add next to $NLIST$; goto $FAIL$
- For ϵ : if $char = \Lambda$ then goto $SUCCESS$ else $FAIL$
- For $E_1 E_2$: code for E_1 ; code for E_2
- For $E_1 \mid E_2$: add E_2 to $CLIST$; code for E_1
- For E_1^* : add $\{ E_1; \text{goto } E_1^* \}$ to $CLIST$; goto next

FAIL:

if $CLIST \neq []$ then pop and goto first element
else { advance $char$; $CLIST = NLIST$; $NLIST = []$ }

Thompson lite

match ::

Regexp \rightarrow [*Regexp*] \rightarrow [*Regexp*] \rightarrow *String* \rightarrow *Bool*

match (*Seq* (*Lit* *c*) *e_k*) *clist* *nlist* *s* | (*head* *s* == *c*) =
resume clist (e_k : nlist) s

match (*Seq* (*Alt* *e₁* *e₂*) *e_k*) *clist* *nlist* *s* =
match (*Seq* *e₁* *e_k*) (*Seq* *e₂* *e_k* : *clist*) *nlist* *s*

...

resume (*c:clist*) *nlist* *s* = *match* *c* *clist* *nlist* *s*

resume [] (*n:nlist*) *s* = *match* *n* *nlist* [] (*tail* *s*)

Parser combinators

expr =

factor \oplus

(**do** *a* \leftarrow *factor*; *eat* '+'; *b* \leftarrow *expr*; return (*Plus a b*))

factor =

(**do** *x* \leftarrow *ident*; return (*Var x*)) \oplus

(**do** *eat* '('; *a* \leftarrow *expr*; *eat* ')'; return *a*)

eat x = (**do** *y* \leftarrow scan; **if** *x* == *y* **then** return () **else** fail)

- Can be implemented with state and backtracking
- Or ...

Claessen [2004]

'Parallel' parser combinators

data *Parser* α =
 Scan (*Token* \rightarrow *Parser* α)
 | *Result* α (*Parser* α)
 | *Fail*

- A parser can: say it wants to know the next token
- or produce a result (and provide alternatives)
- or just fail.

[2] Koen Claessen, "Functional Pearl: Parallel parsing processes," JFP 14, 6 (2004), pp. 741--57.

Alternation – the vital idea

$$\text{Fail} \oplus q = q$$

$$(\text{Result } x \, p') \oplus q = \text{Result } x \, (p' \oplus q)$$

$$(\text{Scan } g) \oplus \text{Fail} = \text{Scan } g$$

$$(\text{Scan } g) \oplus (\text{Result } x \, q') = \text{Result } x \, ((\text{Scan } g) \oplus q')$$

$$(\text{Scan } g) \oplus (\text{Scan } h) = \text{Scan } (\lambda x \rightarrow g \, x \oplus h \, x)$$

- we delay $p \oplus q$ from looking at the next token until both p and q are ready for it.

It's a monad and more

return x = Result x fail

(Result x p) >>= f = Result x (p >>= f)

(Scan g) >>= f = Scan (λ x → g x >>= f)

Fail >>= f = Fail

scan = Scan return

fail = Fail

- These are the operations (*MonadPlus* plus *scan*) needed to write parsers.

Driving a parser

The main program marries the parser state with the stream of input tokens, looking for a result that consumes the whole input.

parse :: *Parser* $\alpha \rightarrow [\textit{Token}] \rightarrow \alpha$

parse (*Scan* *g*) [] = *error* "unexpected EOF"

parse (*Scan* *g*) (*t* : *ts*) = *parse* (*g* *t*) *ts*

parse (*Result* *x* *p*) [] = *x*

parse (*Result* *x* *p*) *ts* = *parse* *p* *ts*

parse *Fail* _ = *error* "syntax error"

- easy to track the latest token for error messages.

Benefits of PPP

- *No backtracking*, so cleans up non-viable alternatives early – simple grammars are usable without transformation or annotation.
- Reads the input *token by token*, so can be made interactive without relying on lazy streams.
Example: prompting for each line of input.
- Will *report first token* that is not part of any legal sentence: one error message for free.
- *Fast enough* to use in practice.

Using continuations

An alternative implementation: each parser take one, two, three continuations.

type *KParser* $\alpha = VCont\ \alpha \rightarrow CCont \rightarrow NCont \rightarrow Answer$

type *VCont* $\alpha = \alpha \rightarrow CCont \rightarrow NCont \rightarrow Answer$

type *CCont* $= NCont \rightarrow Answer$

type *NCont* $= Token \rightarrow CCont \rightarrow Answer$

type *Answer* $= [Token] \rightarrow Value$

- **newtype** is needed all over the place.

A slew of one-liners

The same five operations now have direct definitions.

$$\text{return } x \, k = k \, x$$

$$(p \gg= f) \, k = p \, (\lambda x \rightarrow f \, x \, k)$$

$$\text{fail } k \, ck = ck$$

$$(p \oplus q) \, k \, ck = (p \, k \cdot q \, k) \, ck = p \, k \, \underline{(\lambda nk \rightarrow q \, k \, ck \, nk)}$$

$$\text{scan } k \, ck \, nk = ck \, \underline{(\lambda t \rightarrow nk \, t \cdot k \, t)}$$

Where did that come from?

Define $rep :: Parser\ a \rightarrow KParser\ a$ by

$$rep\ (Scan\ g)\ k\ ck\ nk = ck\ (\lambda\ t \rightarrow nk\ t \cdot k\ t)$$

$$rep\ (Result\ x\ p)\ k\ ck\ nk = k\ x\ (rep\ p\ k\ ck)\ nk$$

$$rep\ Fail\ k\ ck\ nk = ck\ nk$$

Then all else follows!

Deriving bind and plus

In particular, we can prove inductively that

$$\text{rep } (p \gg= f) k = \text{rep } p (\lambda x \rightarrow \text{rep } (f x) k)$$

and

$$\text{rep } (p \oplus q) k ck = \text{rep } p k (\text{rep } q k ck)$$

These justify the new definitions of $\gg=$ and \oplus .

Driving the new parser

$kparse :: KParser\ Value \rightarrow [Token] \rightarrow Value$

$kparse\ p = p\ k_0\ ck_0\ nk_0$

where

$k_0\ x\ ck\ nk\ ts =$

if $ts == []$ **then** x **else** $ck\ nk\ ts$

$ck_0\ nk\ [] = error\ "unexpected\ EOF"$

$ck_0\ nk\ (t:ts) = nk\ t\ ck_0\ ts$

$nk_0\ t\ ck = ck\ nk_0$

Defunctionalising

"Looking for the lambdas", we find that *NConts* are created only by the expression

$$(\lambda t \rightarrow nk\ t \cdot k\ t)$$

(with *k* and *nk* as free variables) and *CConts* only by the expression

$$(\lambda nk \rightarrow q\ k\ ck\ nk)$$

and by promoting *NConts* to *CConts* when scanning.

We can represent both by lists of (ordinary) continuations, with a suitable *resume* function.

Concrete continuations

*scan k clist nlist ts =
 resume clist (k (head ts) : nlist)*

fail k clist nlist = resume clist nlist

(p ⊕ q) k clist nlist = p k (q k : clist) nlist

resume (k : clist) nlist ts = k clist nlist ts

resume [] nlist [] = error "unexpected EOF"

resume [] nlist ts = resume (reverse nlist) [] (tail ts)

- The *reverse* is needed because sometimes we care about the order of results.

Focussing ...

type *KParser* $\alpha =$
 $VCont\ \alpha \rightarrow [Cont] \rightarrow [Cont] \rightarrow [Token] \rightarrow Value$

scan $k\ clist\ nlist\ ts =$
resume $clist\ (k\ (head\ ts) : nlist)\ ts$

$(p \oplus q)\ k\ clist\ nlist =$
 $p\ k\ (q\ k : clist)\ nlist$

resume $(k : clist)\ nlist\ ts = k\ clist\ nlist\ ts$

resume $[]\ (k : nlist)\ ts = k\ nlist\ []\ (tail\ ts)$

Comparing ...

match ::

Regex → [Regex] → [Regex] → String → Bool

*match (Seq (Lit c) e_k) clist nlist s | (head s == c) =
resume clist (e_k : nlist) s*

*match (Seq (Alt e₁ e₂) e_k) clist nlist s =
match (Seq e₁ e_k) (Seq e₂ e_k : clist) nlist s*

resume (c : clist) nlist s = match c clist nlist s

resume [] (n : nlist) s = match n nlist [] (tail s)

Remarks

Discovering this implementation seems to depend on the insight that a normal form for the context of a parser is

$$\text{Scan } (p_1 \oplus \dots \oplus p_k) \oplus (? \gg= g) \oplus (q_m \oplus \dots \oplus q_1)$$

– so that p_1, \dots, p_k and g and q_1, \dots, q_m correspond to *nlist* and k and *clist* respectively.

Can this insight (in general) be replaced by a formal calculation? Why does the (more complicated) free monad implementation seem easier to find?

A zoo of control constructs

Similar remarks apply to:

- Backtracking: [Spivey & Seres; Hinze; Wand & Vaillancourt].
- Coroutine pipelines: [ICFP'17].
- ... and now parser combinators.

Some dreams

- A symbolic reasoning tool that makes higher-order calculations easier (like Mathematica or Alpha), not harder (like any verification tool you know).
- An automated defunctionaliser that helps us to control and visualise the results.